CRYPTOL: High Assurance, Retargetable Crypto Development and Validation

Jeffrey R. Lewis
Galois Connections, Inc
Portland, Oregon

Brad Martin
National Security Agency

October 2003

Abstract

As cryptography becomes more vital to the infrastructure of computing systems, it becomes increasingly vital to be able to rapidly and correctly produce new implementations of cryptographic algorithms. To address these challenges, we introduce a new, formal methods-based approach to the specification and implementation of cryptography, present a number of scenarios of use, an overview of the language, and present part of a specification of the Advanced Encryption Standard.

Introduction

Cryptographic components are increasingly being integrated into hardware and software systems to improve information assurance and security. Because of this, several serious challenges arise: test and verification of systems incorporating cryptography, unambiguous specification of cryptographic algorithms, and the rapid and safe retargeting of cryptographic implementations to new hardware and software platforms.

Cryptol brings a new, formal methods-based approach to cryptography that addresses these challenges. It is a high-level specification language for cryptography that was designed at Galois Connections in consultation with expert cryptographers from the National Security Agency. In Cryptol, cryptographic concepts are expressed directly and formally and in a fashion that is independent of the details of a particular hardware platform.

Cryptol provides significant benefits to:

- Crypto developers targeting a variety of hardware and software platforms
- High-assurance systems developers incorporating embedded cryptographic components
- Cryptographers that explore new cryptographic approaches
- Verification laboratories which use formal models to verify implementations
- Customers of high-assurance systems responsible for validation and test.

These benefits are a result of one's ability to view a single Cryptol specification from a number of perspectives. First Cryptol can be seen as a *language for Cryptography*. Using high-level Cryptol to express the same concepts and idioms as those found in published algorithms, developers can quickly implement pre-existing algorithms or develop new ones. Developers are thereby freed to focus on the cryptography itself, not distracted by machine-level details such as word size. In a complementary way, Cryptol can be seen as providing an *authoritative reference for validation*. To this end, Cryptol is positioned to become the standard language for cryptography. A growing number of both public and non-public algorithms are under development. Standard Cryptol specifications can be used to validate new cryptographic implementations by generating test vectors of user-selectable intermediate values. Taking this line of thinking a bit further, Cryptol may also be viewed as a *framework for verification*. For embedded systems in particular, and for developers of high

assurance applications in general, Cryptol facilitates construction of formal models, providing for an increased level of confidence in the development. Lastly Cryptol provides an exciting *platform for implementation generation*. In this regard it should be stated that Cryptol specifications are inherently portable. Retargeting the deployment platform does not involve recoding the algorithm. Cryptol is intended for use with various platforms, including embedded systems, smart cards, and FPGAs.

Uses of Cryptol

This section presents various scenarios of how Cryptol may be used to address the aforementioned, as well as other challenges in the area of cryptography.

As a formal specification language, Cryptol can be the basis of a standard library of cryptographic specifications. This is useful for current, and evolving standards, but is also very useful for design capture of legacy algorithms that are still deployed, and may need to be redeployed again for compatibility with pre-existing installations. In addition, for current and evolving standards, Cryptol can provide the basis for complete design capture of highly parameterized algorithms. Many algorithms are parameterized on things like block size, key size, number of rounds, etc, but in practice are only specified for certain fixed standard sizes. Cryptol was designed to be well suited to capturing the complete parameterized design.

Another interesting area of application for Cryptol as a specification language is towards the specification of new modes. The number of cryptographic algorithms in use is relatively stable, but the modes of use of them are still an evolving area. Cryptol can be an excellent design tool for new modes, as well as provide the framework for libraries of modes.

Libraries of Cryptol specifications are also useful for validating implementations of crypto algorithms. Such libraries can be used as a golden reference for test vectors. Not only can it be used for arbitrary input-output vectors, it can also be used to generate vectors for any sub-part of an algorithm. The specification could be used as the basis for generation of large sets of random test vectors on-demand—for any part of the algorithm. One scenario is to imagine hooking up crypto implementations to a Cryptol test harness, which would feed large sets of fixed and random test vectors to the implementation, instead of rely on testing just via a fixed small set of vectors.

Taking this one step further, Cryptol could be used as a basis for machine assisted verification. Due to the intentionally chaotic nature of cryptography (i.e. it intentionally spreads information over as wide an area as possible), it would be hopeless to apply techniques such as model checking in a naïve fashion, since the state space would rapidly blow up. However, we can easily imagine applying such brute force verification techniques to subparts of an algorithm. Fortunately, cryptographic algorithms are fairly stylized: there's a brief initialization phase, a number of rounds, and a brief finalization phase. We can easily apply brute force verification to the initialization and finalization. But we can also apply brute force model checking to verify the body of the round function without having to actually iterate it. Then, for any given algorithm, we can use a stock argument based on induction and composition to combine the verification steps on the parts of the algorithm into a verification of the correctness of the entire algorithm.

Cryptol is perhaps most useful as a platform for implementation generation. The total number of cryptographic algorithms out in the world is relatively small, and will most likely continue to be so. However, the total number of implementations of cryptographic algorithms is growing rapidly, and will continue to do so. As the use of cryptography becomes more and more ubiquitous, the need to deploy it on more and more platforms will grow. Many of the most interesting platforms for cryptography are on embedded processors and other specialized hardware that have a wide variety of requirements to satisfy, and thus require a wide variety of implementations.

There are a number of ways that Cryptol can be used in implementation generation. The most straightforward

is to use Cryptol to generate reference implementations in a variety of industrial languages, such as Java, or C# for software implementations, or VHDL for hardware. The expectation is that the generated code would be designed for readability instead of efficiency.

Another way is to use Cryptol to interface with existing implementations. For example, you may have an existing efficient implementation of a block algorithm, and use Cryptol to generate the code for a new mode. In this case, the Cryptol generated code need not be highly optimized since it will be responsible for a tiny portion of the runtime.

Another mode of use would be to use Cryptol to generate moderately efficient code, which is then hand optimized in the critical portions to attain the needed efficiency. Alternately, you might link Cryptol-generated code to a library of highly tuned core functions.

But perhaps the biggest payoff in code generation from Cryptol specification is targeting directly to hardware. There are two main benefits that Cryptol provides here. The first is that Cryptol specifications themselves are inherently platform independent. They are not contingent on details such as word size of some underlying architecture, and thus can be easily mapped to the particular requirements of a target platform. The second is that Cryptol specifications avoid unnecessary sequentiality, thus they are well suited to taking advantage of the highly parallel nature of hardware.

Current Practice in Cryptography

Current practice in cryptography is for algorithms to be specified in a published paper using a mixture of English text and pseudo code. Associated with the paper specification is usually a reference implementation written in the C language. Unfortunately, neither English, pseudo code or C code are ideal as a basis for a specification.

There are several problems with English and pseudo code specifications. First, they are often incomplete and/or ambiguous, and since there's no practical way to machine check them, there's no easy way to determine whether the specification is complete and unambiguous. This in itself makes such a specification inadequate as a basis for verification. Further, a paper specification is not executable. This makes validation based on it very problematic. There's usually only a handful of test vectors supplied—if more are needed, the only recourse is to attempt to calculate additional vectors by hand. The pseudo code used is also typically a Pascal-like procedural language. This has two problems. First, a procedural specification will invariably obscure the underlying mathematics inherent in a cryptographic algorithm. Second, a procedural specification will needlessly enforce a sequential order upon the algorithm. This means that the specification will be inappropriate as a basis for implementation on a highly parallel platform.

There are also a number of problems with using C as the specification language. A C implementation is certainly executable, but unless extreme care is taken, it correctness depends upon what platform the implementation is executed on. The C language is also far too low-level. An implementation has to concern itself with various details about the platform on which it is executed, such as the word size of the platform. The code that implements 4-bit vectors will be radically different from the code that implements 48-bit vectors, and is usually forced to be highly dependent on conditional compilation constructs or awkward preprocessor macros. Further, a C implementation has to concern itself with memory allocation, memory organization and pointer manipulation details that are ripe sources of errors in C code.

A Domain Specific Language for Cryptography

Whenever you have a specialized particular application area, such as cryptography, there is usually a significant gap between the concepts fundamental to that application area, and the concepts available in a traditional programming language. This gap causes a

tension whereby something is lost: usually either the clarity and expressiveness of programs or the efficiency of implementation. This tension also forces those that implement applications in this area to be experts both in the application domain, as well as experts in programming. Domain Specific Languages are languages tailored to the needs of a particular application area that bridge the gap between the application concepts and programming concepts. A Domain Specific Language allows experts in the application domain to express their ideas directly without needing to become expert programmers.

Cryptography is a fundamentally mathematical discipline with its own specialized idioms. At the same time the implementation of cryptography must be absolutely correct and highly efficient in order to be practical and its use accepted. This sort of intersection of specialized needs with demanding implementation requirements lends itself well to the use of a Domain Specific Language.

Introduction to Cryptol

Cryptol was designed to meet the challenges facing crypto implementation. As a language for cryptography, it was designed with feedback from expert cryptographers at the National Security Agency, and thus naturally speaks the language of cryptographers. As a platform for validation and verification, it is a formal language, and thus is designed to be complete and unambiguous. As a platform for generation, Cryptol is a declarative language that is platform neutral.

The Cryptol language was initially designed to target block symmetric cryptographic algorithms. During the design phase, the five finalists for the Advanced Encryption Standard (MARS, Serpent, TwoFish, RC6, and Rijndael) were studied as good examples of state of the art cryptographic algorithms. In addition, DES was studied as an example of current practice. In studying these various algorithms, the idea was to identify what the algorithms had in common, as well as what differences occurred among them. This process identifies the idioms and concepts of a domain, as well as the range of expression that is needed to capture designs.

The data in cryptographic algorithms is typically vectors of bits of varying sizes, usually ranging from 4 bits upwards to 128 bits, with 8 bit and 32 bit being common sizes. Bit vectors are grouped together in various ways (such as a 2D matrix of bits) to form blocks, where a block is the unit of encryption, and in the Advanced Encryption Standard consists of 128 bits. To encrypt arbitrary amounts of data, a block algorithm is iterated over a stream of blocks.

Another form of data in crypto algorithms is lookup tables, also known as “substitution boxes”, or S-Boxes for short. These are relatively small tables, e.g. a table that maps 4-bit vectors to 8-bit vectors.

Cryptol uses the simple uniform concept of a sequence to express how data is organized in a crypto algorithm. Bit vectors are sequences of bits, matrices, tables and blocks are sequences of sequences, and streams are just sequences of blocks.

Literal sequences in Cryptol are written using brackets surrounding the elements, and spaces separating them. Sequences are indexed starting with zero, and are written left-to-right in increasing index order. Bits are written using the constants **`True`** and **`False`**. Thus, the following is a 7 element sequence of bits:

```
[True False False True False True True]
```

Numbers in Cryptol are represented by bit vectors, and, as is typical with crypto algorithms, explicitly use modulus arithmethic, with the modulus based on the size of the vector. For example, an 8-bit vector would support arithmetic modulo 2^8. Numeric literals can be written in the usual fashion, using the C convention for expressing literals in a base other than 10, i.e. **`0x10`** is a hexadecimal literal whose value is sixteen. Numbers are encoded as sequences of bits using the little-endian convention. Thus, the above sequence of bits may also be written:

```
0x69
```

In addition to standard arithmetic, Cryptol also supports polynomial arithmetic, which occurs in some advanced cryptographic algorithms, such as AES and TwoFish. Addition, multiplication, division, and modulus are all supported over polynomials. Polynomials are written in a fashion suggestive of the mathematical notation. For example, the polynomial x^7+x^5+x+1 is written in Cryptol as:

```
<| x^7 + x^5 + x + 1 |>
```

Polynomials are represented as a sequence of the coefficients, with the coefficient for x^n being the nth element of the sequence. Commonly, the coefficients are simply bits. For example, the polynomial written above is represented as the sequence:

```
[True True False False
 False True False True]
```

There is a rich set of operators for manipulating sequences, including basic ones like sequence concatenation (#), and sequence indexing (@).

In addition to these various operators on sequences, *sequence comprehensions* allow element-wise specification of sequences. A sequence comprehension has two parts: a defining expression, and a list of generating sequences. The generating sequences provide elements and the defining expression says how to combine those elements into an element of the resulting sequence. Here's an example:

```
[| 2*x + y || x <- xs || y <- ys |]
```

The expression **2*x + y** is the defining expression, and the generating sequences are specified by **x <- xs** and **y <- ys**, where **xs** and **ys** are the names of two sequences, and **x** and **y** represent single elements drawn from each sequence. The sequence that results from this comprehension is defined as follows: the *i*th element of the result sequence is defined as the value of the expression **2*x + y**, where **x** is equal to the *i*th element of **xs**, and **y** is equal to the *i*th element of **ys**. The length of the new sequence is the minimum of the lengths of the generating sequences. For example, given the sequences:

```
xs = [1 2 3]
ys = [5 6 7 8]
```

The resulting sequence is:

```
[7 10 13]
```

Control flow in cryptographic algorithms is typically quite straightforward as data-dependent control is avoided to prevent timing attacks. Most control flow consists of simple iteration, and is written as for-loops in pseudo code specifications. Unfortunately, for-loops encode sequentiality, even when that sequentiality is not inherent in the specification. Cryptol takes a declarative approach: you specify a sequence of the intermediate values leading to a final value (in the style of a recurrence relation), instead of the sequentially imperative style of specifying the steps you would take to arrive at the final value. The declarative approach has an added benefit: it automatically provides a handle on all the intermediate values. This is invaluable when using Cryptol to generate test vectors.

Recurrence relations are specified in Cryptol as recursive sequences. For example, the following function sums up the elements of its argument sequence by specifying all the intermediate sums leading up to the final sum, and taking the last element of that sequence to get the result.

```
sum xs = last ys
  where
    ys = [0] #
         [| x + y || x <- xs || y <- ys
|];
```

The final aspect of Cryptol that we need to touch on in this brief introduction is the use of types. Types in Cryptol express the size and shape of data. Consider the sequence

```
zs = [[0x7a 0x1b] [0x26 0x5c] [0xb4
0x11]];
```

This is a sequence of 3 elements, each of which is a sequence of 2 elements, each of which is a sequence of 8 bits. We write this in Cryptol as: **`[3][2][8]Bit`**. The size of each sequence is given, wrapped in brackets, from outermost to innermost. The innermost type defaults to **`Bit`** and can be elided for convenience.

Cryptol is a strongly typed language: every expression and definition must have a valid type, and every value and function must be used in a manner that is consistent with its type.

The user may also supply type annotations on definitions and expressions as a form of documentation that is checked by Cryptol for consistency. Type annotations on definitions, also called type signatures, as written as a name, followed by a colon, and the type that that name is to adhere to. For example, a valid type signature for **`zs`** defined above is:

```
zs : [3][2][8];
```

AES in Cryptol

In this section, we present portions of a specification of the Advanced Encryption Standard (AES) in Cryptol.

The first thing to define is the basic parameters of the algorithm. The Rijndael algorithm, upon which AES is defined, has three parameters: **`Nb`**, **`Nk`**, and **`Nr`**. These three parameters specify the size (in bits) of the input block divided by 32, the size of the key divided by 32, and the number of rounds. The division by 32 has to do with the fact that AES internally represents both the block and the key as two dimensional matrices of bytes, where the number of rows is fixed at four, and the number of columns is specified by **`Nb`** and **`Nk`** respectively (4*8=32). The AES is defined as the instance where **`Nb`** is 4, **`Nk`** is either 4, 5, or 6, and **`Nr`** is defined as the largest of **`Nb`** and **`Nk`** plus 6.

```
Nb = 4;
Nk = 4;
Nr = max(Nb, Nk) + 6;
```

There's one more preliminary step—defining some abbreviations that will be convenient later. The first is an abbreviation for the type of the state, the two-dimensional matrix of bytes that is the internal presentation of the block of data being encrypted/decrypted. The second is an abbreviation for the type of the expanded key material.

```
State = [4][Nb][8]
Xkey = (State, [Nr-1]State, State)
```

`Xkey` is a triple consisting of initial key material, the key material for the middle rounds, and the key material for the final round. We can now write type signatures for the inferface to the algorithm:

```
keySchedule : [4*Nk][8] -> Xkey
encrypt : (Xkey,[4*Nb][8]) -> [4*Nb][8]
decrypt : (Xkey,[4*Nb][8]) -> [4*Nb][8]
```

The first signature says that **`keySchedule`** is a function that takes **`4*Nk`** bytes, and returns the expanded key material (***`Xkey`***). The functions encrypt and decrypt take the expanded key, and **`4*Nb`** bytes, and returns the same. We now look at the top-level encryption function.

```
encrypt (XK, PT) =
  unstripe (Rounds (State, XK))
  where State = stripe PT;
```

As you can see, local definitions can be introduced using **`where`** clauses. The top-level call is basically a call to the **`Rounds`** function wrapped with calls to **`stripe`** and **`unstripe`** respectively. The stripe and unstripe functions convert from and to the input, which is a flat sequence of bytes into the two-dimensional internal form of the *state*.

```
stripe : [Nb*4][8] -> [4][Nb][8];
stripe block = transpose (split block);
```

```
unstripe : [4][Nb][8] -> [Nb*4][8];
unstripe state = join (transpose state);
```

The split operator takes a sequence and splits it into a sequence of sequences—where the amount of elements (and the number of sub-elements in each) it splits the input into is guided by the type signature. Then, because the input is mapped onto the state in column-major fashion, the split sequence is transposed. So, first the **[Nb*4][8]** sequence is split into a **[Nb][4][8]** sequence, then it is transposed into column-major format to a **[4][Nb][8]** sequence. The **unstripe** function is reverse analogous—first transpose, the use **join**, to combine the two-dimensional sequence of bytes end-to-end into a one-dimensional sequence.

The Rounds function specifies the round structure of the algorithm. First, the initial key material is added into the initial state to make the starting point for the rounds iteration (**rnd0**). Then, we define the sequence of round intermediate values using a recursive sequence definition (**rnds**). Lastly, the final round is called on the last of the round intermediate values with the final key (the final round in AES is a variant of the previous rounds). The body of the rounds iteration is simply a call to the **Round** function using the current state, and the round keys for that round.

```
Rounds (State,
        (initialKey, rndKeys, finalKey))
=
  final
  where {
    rnd0 = AddRoundKey(State,initialKey);
    rnds = [rnd0] # [| Round (state, key)
                    || state <- rnds
                    || key <- rndKeys |];
    final =
      FinalRound (last rnds, finalKey);
  };
```

The **Round** function is the heart of the algorithm. It takes the current state, and a round key, and takes the state through four transformations, returning the final result.

```
Round : (State, State) -> State;
Round (State, RoundKey) = State4
  where {
    State1 = ByteSub State;
    State2 = ShiftRow State1;
    State3 = MixColumn State2;
    State4 =
      AddRoundKey (State3, RoundKey);
  };
```

The **BytesSub** function is a ***State*** to ***State*** transforming function, as the type signature indicates. It is defined as a nested comprehension—for each row in the state, it produces a new row, where each element has had the **sbox** function applied to it. The effect is simply an element-wise application of the sbox function to each byte of the state.

```
ByteSub : State -> State;
ByteSub state =
  [| [| sbox x || x <- row |]
  || row <- state |];
```

The **ShiftRow** function is also a **State** to **State** transforming function. For each row of the state, that row is circularly shifted a certain number of positions to the left. The amount of the shift depends on which row, so the comprehension has an additional generator: **i<-[0 ..]** that generates an increasing index counter starting with 0. This index is used in the call to the **shift** function, which calculates the actual amount of shift. The **<<<** operator performs a circular shift of the top-level elements of its left-hand argument sequence.

```
ShiftRow : State -> State;
ShiftRow state =
  [| row <<< shift (Nb, i)
  || row <- state
  || i <- [ 0 .. ] |];

shift : ([8], [4]) -> [4];
shift (nb, i) = j
```

```
  where j = if (i < 2) | (nb < 8)
            then i else i + 1;
```

The **MixColumn** function is again a **State** to **State** transformer. It operates on the state in a column major fashion—hence the state is first transposed, then we perform the **multCol** operation (which we won't look at here) on each column, and transpose the result.

```
MixColumn : [4][Nb][8] -> [4][Nb][8];
MixColumn state =
  transpose [| multCol (cx, col)
             || col <- transpose state
             |];
```

It's worth noting that the above is a specification—not an imperative as to how to implement the specification. If this code were compiled, an optimizing compiler has enough information to turn the transpose operations into an efficient re-indexing of the state, instead of a costly pair of transpose operations.

The final round transformation adds the round key material to the state by bit-wise exclusive-or. In Cryptol, this is easily expressed using the exclusive-or operator **^**. The exclusive-or operator, like all of the Boolean operators, applies to arbitrary types by extending bit-wise in the natural fashion.

```
AddRoundKey : (State, State) -> State;
AddRoundKey (State, Key) = State ^ Key;
```

Status of Cryptol

Galois Connections has currently implemented the Cryptol™ development tool. It provides an interpreter for Cryptol, a tracing and debugging facility that is useful for generating test vectors and intermediate values for any named subpart of a specification. It also provides a compiler that generates C code.

Currently under development is support for automated verification using Binary Decision Diagrams (BDDs). In addition, an optimized version of the C compiler is also underway.

In the near future, we expect to begin work on a compiler to target Cryptol specifications straight down to FPGAs, which are reconfigurable hardware devices.

Conclusion

There are a number of challenges in implementing cryptography efficiently and correctly. The highly specialized nature of cryptography, and the many requirements on implementations under a wide variety of architectures offers a ripe opportunity for a specialized language and tools. Cryptol was designed to meet these needs. We have presented a number of ways in which Cryptol may be used, given a flavor of what the language looks like, and shown what the current Advanced Encryption Standard looks like in Cryptol.